Programare dinamică 1

Prezentare generală

Programarea dinamică este o metodă de elaborare a algoritmilor care se aplică în general problemelor pentru care se cere determinarea unui optim în urma adoptării unor decizii.

Nu există un criteriu pe baza căruia să identificăm cu siguranţă o problemă pentru rezolvarea căreia trebuie să utilizăm metoda programării dinamice, dar putem formula două proprietăţi care sugerează o soluţie prin programare dinamică.

- *Substructură optimală*

Problema dată poate fi descompusă în subprobleme şi soluţia optimă a problemei depinde de soluţiile optime ale subproblemelor sale.

Acest criteriu nu indică neapărat o soluţie prin programare dinamică, ar putea fi şi un indiciu că se poate aplica metoda *Greedy* sau metoda „*Divide et Impera*".

- *Subprobleme superpozabile*

Subproblemele problemei date nu sunt independente, ci se suprapun.

Datorită faptului că subproblemele problemei date se suprapun, deducem că o abordare prin metoda „*Divide et Impera*" ar fi dezastruoasă din punctul de vedere al timpului de execuţie (datorită faptului că problemele se suprapun se ajunge la rezolvarea repetată a aceleiaşi subprobleme). Prin urmare, vom rezolva subproblemele o singură, dată, reţinând rezultatele într-o structură de date suplimentară (de obicei un tablou).

Rezolvarea unei probleme prin programare dinamică presupune următorii paşi:

1. Se identifică subproblemele problemei date.
2. Se alege o structură de date capabilă să reţină soluţiile subproblemelor.
3. Se caracterizează substructura optimală a problemei printr-o relaţie de recurenţă.
4. Pentru a determina soluţia optimă, se rezolvă relaţia de recurenţă în mod *bottom-up* (se rezolvă subproblemele în ordinea crescătoare a dimensiunii lor).

Aplicaţii

Codificare optimală

Fie un text de lungime maximă 100, ce conţine doar litere. Textul poate fi codificat, înlocuind apariţiile consecutive ale subsecvenţelor sale cu subsecvenţa respectivă urmată de numărul său de apariţii.

Exemplu
Textul `T="aaacaaacbbdefdef"` poate fi codificat: `"a3c1a3c1b2def2"` dar şi `"aaac2b2def2"`. Evident, cel de-al doilea mod de codificare este mai scurt, deci mai convenabil.

Scrieţi un program care să codifice optimal un text dat (codul rezultat să fie de lungime minimă).

Soluţie

1. Spaţiul subproblemelor problemei iniţiale este format din determinarea unei codificări optimale pentru caracterele din text de la `i` până la `j` (`T[i..j]`), `0≤i≤j<n`, unde `n` este lungimea textului. Evident, subproblemele se suprapun. De exemplu, determinarea codificării optime pentru `T[i..j]`, necesită determinarea unor codificări optime pentru `T[i..k]` şi pentru `T[k+1..j]`, `k∈{0,1,..., j-1}`.
2. Pentru a memora soluţiile subproblemelor, vom utiliza o matrice `Lg`, pătratică de ordin `n`, având semnificaţia `Lg[i][j]` = lungimea codificării optime pentru `T[i..j]`; `0≤i≤j<n`. Observaţi că este utilizată numai jumătatea de deasupra diagonalei principale.
3. Problema are substructură optimală, caracterizată de următoarea relaţie de recurenţă:

Orice caracter x se codifică pe două poziţii (x1), deci:

```
Lg[i][i]=2, ∀i∈{0,1,..., n-1}
Lg[i][j]=min {j-i+2, minI, minII}
```

unde:

- `j-i+2` provine din codificarea întregului şir, urmat de `1`
- $\texttt{minI}=\min_{k=i,j-1}\{\texttt{Lg[i][k]+Lg[k+1][j]}\}$

  Codificăm optimal `T[i..j]` concatenând codificările optimale ale şirurilor `T[i..k]` şi `T[k+1..j]` (poziţia `k∈{i, i+1,...,j-1}` se alege în mod optimal)

- `minII=min{strlen(s)+NrCifre(k)}`, unde `s` subşir al lui `T[i..j]` astfel încât `T[i..j]=ss..s` (de `k` ori). Deci `T[i..j]` se codifică `sk`.

Deoarece jumătatea de sub diagonala principală din matricea l este neutilizată, pentru a putea reconstitui soluţia optimă, o vom utiliza astfel:

- `Lg[j][i]=0`, dacă `Lg[i][j]=j-i+2`
- `Lg[j][i]=k`, unde k este poziţia de splitare a textului pentru `Lg[i][j]=minI`
- `Lg[j][i]=-k`, unde `k` este lungimea subşirului `s` care se repetă, pentru `T[i][j]=minII`.

4. Rezolvarea relaţiei de recurenţă:

```
for (int i=0; i<n; i++) Lg[i][i]=2;
for (int d=2; d<=n; d++)   //codific subsiruri de lungime d
    for (i=0; i<=n-d; i++) //care incep la pozitia i
        {int j=i+d-1;      //si se termina la pozitia j
         //cazul I- codificarea este intregul sir urmat de 1
        Lg[i][j]=d+1; Lg[j][i]=0;
        //cazul II
        for (int k=i; k<j; k++)
            if (Lg[i][j]>Lg[i][k]+Lg[k+1][j])
               {Lg[i][j]=Lg[i][k]+Lg[k+1][j];
                Lg[j][i]=k;}// pozitia optima de splitare
        //cazul III
        for (k=1; k<=d/2; k++)
            if (d%k==0) //determin un subsir de lungime k
               if (SeRepeta(i,j,k))
                  {Lg[i][j]=k+NrCifre(d/k);
                   Lg[j][i]=-k; //retin lungimea subsirului
                   break;}
        }
```

Observaţi că am utilizat funcţia NrCifre() care determină numărul de cifre dintr-un număr natural, precum şi funcţia SeRepeta().

```
int SeRepeta (int i, int j, int k)
{/*verifica daca sirul T[i..j] este format prin concatenarea succesiva a lui
T[i..i+k-1] */
char s[DimMax], s1[DimMax];
strncpy(s,T+i,k); s[k]=NULL; s1[k]=NULL;
for (int h=i+k; h<=j; h+=k)
    {strncpy(s1,T+h, k);
     if (strcmp(s,s1)) return 0;}
return 1;}
```

Afişarea soluţiei optime o vom realiza prin apelarea funcţiei recursive Afisare(0,n-1):

```
ofstream f("cod.out");
void Afisare(int i, int j)
{
if (i==j) cout<<T[i]<<1;     //un singur caracter
   else
   {int k=Lg[j][i];
    char s[DimMax];
    if (!k)                  // cazul I
       {strncpy(s,T+i,j-i+1); s[j-i+1]=NULL;
        cout<<s<<1; }
      else
      if (k>0)               //cazul II
         {Afisare (i, k);
          Afisare (k+1,j);}
         else                //cazul III
         {strncpy(s,T+i,-k); s[-k]=NULL;
          cout<<s<<(j-i+1)/(-k);}
    }
}
```

rv (campion)

Timp maxim de execuţie/test:	**0.1 secunde**
Memorie totală disponibilă/stivă:	**2 MB/1 MB**

Ana şi Bogdan joacă un nou joc, numit R&V. Jocul are două table de joc: una roşie şi una verde. Pe tabla roşie este iniţial plasat un şir de jetoane, pe fiecare jeton fiind scrisă o literă mică a alfabetului englez. Tabla verde este iniţial goală.
Ana şi Bogdan execută mutări pe rând, Ana fiind prima la mutare. La o mutare, jucătorul care este la rând trebuie să ia un jeton de la unul dintre capetele şirului aflat pe tabla roşie şi să îl plaseze pe tabla verde, la sfârşitul şirului de jetoane de pe această tablă. Jocul continuă până când toate jetoanele se află pe tabla verde.
Scopul Anei este ca şirul obţinut pe tabla verde să fie cât mai mare din punct de vedere lexicografic.
Scopul lui Bogdan este ca şirul obţinut pe tabla verde să fie cât mai mic din punct de vedere lexicografic.

Cerinţă

Scrieţi un program care să determine şirul obţinut pe tabla verde la finalul jocului, în ipoteza că ambii jucători joacă optimal.

Date de intrare

Fişierul de intrare rv.in conţine pe prima linie un şir de litere mici ale alfabetului englez, reprezentând în ordine literele de pe jetoanele aflate pe tabla roşie la începutul jocului.

Date de ieşire

Fişierul de ieşire rv.out va conţine o singură linie pe care vor fi scrise în ordine literele scrise pe jetoanele de pe tabla verde la finalul jocului, în ipoteza că ambii jucători joacă optimal.

Restricţii

- Pe tabla roşie se află iniţial cel mult 100 de jetoane.
- Fie $x_1x_2...x_n$ si $y_1y_2...y_n$ doua siruri. Spunem ca x este mai mic decat y din punct de vedere lexicografic daca exista un indice k (1<=k<=n) astfel incat $x_i=y_i$ (pentru orice 1<=i<k) si $x_k<y_k$.
- Dacă un jucător poate juca optimal selectând atât din stânga, cât şi din dreapta, va alege stânga.

Exemple

rv.in	**rv.out**
abcde	eadbc

rv.in	**rv.out**
aaaca	aacaa

Soluţie

Vom memora şirul dat în `s` (`n` fiind lungimea şirului).

Subproblemă:

Să se determine şirul care se obţine pe tabla verde (în ipoteza ca ambii jucatori joaca optimal) dacă pe tabla roşie se află subsecvenţa `s[i..j]` (`0<=i<=j<n`)

Vom reţine rezultatele subproblemelor în tabloul `rez`.

`char rez[MAX][MAX][MAX];`

`rez [i][j]`=şirul obtinut pe tabla verde dacă pe tabla roşie se află subsecvenţa `s[i..j]` (`0<=i<=j<n`)

Relatia de recurenta

`rez[i][i]=s[i];`

Dacă la mutare este primul jucător:

`rez[i][j]=max{s[i]+rez[i+1][j], s[j]+rez[i][j-1]}`

Daca la mutare este al doilea jucator:

`rez[i][j]=min{s[i]+rez[i+1][j], s[j]+rez[i][j-1]}`

(unde cu + am notat operaţia de concatenare)

Rezultatul problemei va fi memorat în `rez[0][n-1]`

```
#include <stdio.h>
#include <string.h>
#define MAX 104
int n;
char s[MAX];
char rez[MAX][MAX][MAX];
//rez [i][j]=sirul obtinut pe tabla verde daca pe tabla rosie se afla secventa de
litere de la i la j

int cine_muta(int p, int q)
{
 if ((q - p + 1) % 2 == n % 2) return 1;
 return 2;
}

int main()
{
  int i, j, d, cine, test;
  char s1[MAX], s2[MAX];
  freopen("rv.in", "rt", stdin);
  freopen("rv.out", "wt", stdout);
  scanf("%s",s);
  n = strlen(s);
  for (i=0; i<n; i++)
      {rez[i][i][0]=s[i]; rez[i][i][1]=0;}
  for (d=2; d<=n; d++)
      for (i=0; i<n; i++)
          {j=i+d-1;
           cine = cine_muta(i, j);
           s1[0]=s[i]; s1[1]=0; strcat(s1,rez[i+1][j]);
           s2[0]=s[j]; s2[1]=0; strcat(s2,rez[i][j-1]);
           test=strcmp(s1,s2);
```

```
            if (cine == 1)
               if (test<=0) strcpy(rez[i][j],s2);
                  else strcpy(rez[i][j],s1);
               else
               if (test<=0) strcpy(rez[i][j],s1);
                  else strcpy(rez[i][j],s2);
            }
  printf("%s\n", rez[0][n-1]) ;
  return 0;
}
```

an (pbinfo) 100 puncte

Ana şi Bogdan au inventat încă un joc. Jocul are jetoane, albe şi negre, care iniţial se aşază într-un teanc, într-o ordine oarecare. Numim *configuraţie* succesiunea culorilor tuturor jetoanelor din teanc (în ordine, începând din vârful teancului). Un jeton alb va fi codificat prin litera A, iar un jeton negru prin litera N.

La o mutare un jucător poate lua din vârful teancului oricâte jetoane consecutive (dar cel puţin un jeton), cu condiţia ca toate jetoanele luate să aibă aceeaşi culoare. Jucătorii mută alternativ, prima la mutare fiind Ana. Jocul va fi câştigat de jucătorul care ia ultimul jeton.

Spunem că un jucător are *strategie sigură de câştig* dacă el, urmând această strategie, câştigă jocul, indiferent care sunt mutările celuilalt jucător.

Cerinţă

Scrieţi un program care citeşte T configuraţii şi determină pentru fiecare dintre cele T configuraţii dacă Ana are strategie sigură de câştig.

Date de intrare

Fişierul de intrare an.in conţine pe prima linie un număr natural T care reprezintă numărul de configuraţii. Pe următoarele T linii sunt scrise cele T configuraţii, câte o configuraţie pe o linie, sub forma unei succesiuni de litere din mulţimea {A, N}.

Date de ieşire

Fişierul de ieşire an.out va conţine T linii. Pe a i-a linie va fi scrisă valoarea 1 dacă Ana are strategie sigură de câştig pentru cea de a i-a configuraţie din fişierul de intrare, respectiv valoarea 0 în caz contrar.

Restricţii şi precizări

- $1 < T \leq 50$
- 0 < numărul de jetoane din orice configuraţie ≤ 10000

Exemple

an.in	an.out	**Explicaţie**
3 A AN NNNAA	1 0 1	Prima configuraţie: există un singur jeton, îl ia Ana şi câştigă. A doua configuraţie: Ana este obligată să ia primul jeton, Bogdan îl va lua pe cel de al doilea şi câştigă Bogdan. A treia configuraţie: Ana ia primele două jetoane. Bogdan va fi obligat să ia al treilea jeton. Ana ia ultimele două jetoane şi câştigă.

Timp maxim de execuţie/test: 0.1 secunde
Memorie totală disponibilă 2 MB din care 2 MB pentru stivă

Descrierea soluţiei problemei an

Vom citi succesiv cele T configuraţii. Pentru fiecare configuraţie vom determina dacă primul jucător are strategie sigură de câştig.

Subproblemă

Să se determine dacă Ana are strategie sigură de câştig, dacă atunci când este la mutare în stivă sunt jetoanele i, i+1, ..., n-1 (0<=i<n).

Vom reţine soluţiile subproblemelor într-un vector

`c[i]=1`, dacă Ana are strategie sigură de câştig pentru jetoanele i, i+1, ..., n sau 0 în caz contrar.

Relaţia de recurenţă

`c[n-1]=1`

Să considerăm că la începutul configuraţiei există `nr` jetoane de aceeaşi culoare (`nr>1`):

`XX...XO...`

În acest caz Ana are strategie sigură de câştig, pentru că poate proceda astfel:

Dacă pentru configuraţia care începe cu O Bogdan ar avea strategie sigură de câştig, atunci Ana ia nr-1 jetoane, îl obligă astfel pe Bogdan să ia ultimul jeton X, iar Ana ajunge într-o poziţie cu strategie sigură de câştig. Dacă pentru configuraţia care începe cu O Bogdan nu are strategie sigură de câştig, atunci Ana ia toate cele nr jetoane. Dacă la începutul configuraţiei este un singur jeton de o culoare, după care urmează un jeton de altă culoare: XO... În acest caz Ana este obligată să ia primul jeton, deci dacă pentru configuraţia care începe cu O Bogdan are strategie sigură de câştig atunci Ana nu va avea şi invers.

`c[i]=1-c[i+1], dacă s[i]!=s[i+1]`

`c[i]=1, dacă s[i]==s[i+1]`

```
#include <fstream>
#include <cstring>
#define NMAX 10004
using namespace std;
ifstream fin("an.in");
ofstream fout("an.out");
int T, n;
char s[NMAX];
bool c[NMAX];
int main()
{int i, j;
 fin>>T;
 for (j=0; j<T; j++)
     {fin>>s; n=strlen(s);
      c[n-1]=1;
      for (i=n-2; i>=0; i--)
           if (s[i]!=s[i+1]) c[i]=1-c[i+1];
               else c[i]=1;
      fout<<c[0]<<'\n';
     }
 fout.close(); return 0;
}
```

Problema 5

Două persoane X şi Y joacă următorul joc. Dintr-o grămadă de `n` obiecte, cei doi jucători iau pe rând maximum `m` obiecte. Iniţial mută X şi va câştiga dacă la final are un număr par de obiecte.

Se cere:

a) Să se determine, dacă jucătorul X are strategie sigură de câştig.

b) În caz afirmativ, să se programeze mutările lui X, cele ale lui Y fiind citite de la intrare.

Soluţie

O stare a jocului este caracterizată prin următoarele elemente:

1. numărul de obiecte rămase în grămadă
2. dacă numărul obiectelor pe care le are X este par;
3. cine este la mutare.

Putem codifica informaţiile 2 şi 3 astfel:

0 (00): X are un număr par de obiecte şi este la mutare

1 (01):X are un număr par de obiecte şi nu este la mutare

2 (10): X are un număr impar de obiecte şi este la mutare

3(11): X are un număr impar de obiecte şi nu este la mutare

Deci stările pot fi codificate printr-o pereche de două valori: `nr` de obiecte din grămadă (`0`, `1`, `2`, …, `n`) şi codul (`0`,`1`,`2`,`3`).

În fiecare astfel de stare jucătorul poate să facă maximum `m` mutări diferite. Fiecare mutare va trimite jocul într-o nouă stare în care va juca partenerul.

Subproblemă

Să se determine dacă X are strategie sigură de câştig ştiind că în grămadă sunt `i` obiecte iar codul este `c`.

Vom reţine rezultatele subproblemelor într-o matrice `S[n+1][4]`

`S[i][c]=1`, dacă X are strategie sigură de câştig atunci când în grămadă sunt `i` obiecte, iar codul este `c` şi `0` în caz contrar.

`i=0` este starea de final a jocului

`S[0][0]=S[0][1]=1; S[0][2]=S[0][3]=0.`

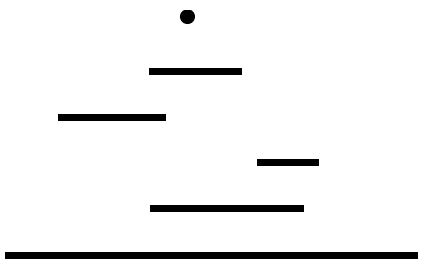
`S[i][c]`

Dacă `c==0` sau `c==2` (adică X este la mutare) `S[i][c]` va fi `1`, dacă există `j` (`1<=j<=m`) astfel încât `S[i-j][1 sau 3]` este `0`

Dacă `c==1` sau `c==3` (adică Y este la mutare) `S[i][c]` va fi `0` dacă există există `j` (`1<=j<=m`) astfel încât `S[i-j][0 sau 2]` este `0` şi `1` în caz contrar

Fall (campion)

Considerăm un joc bazat pe un dispozitiv ca în imaginea de mai jos:



Dispozitivul constă dintr-o mulţime de platforme orizontale de diferite lungimi, plasate la diferite înălţimi. Platforma cea mai joasă este podeaua (este plasată la înălţimea 0 şi are lungime infinită).
Dintr-o poziţie specificată, este lăsată să cadă o minge, la momentul 0. Mingea cade cu viteză constantă de 1 metru pe secundă. Când mingea ajunge pe o platformă, începe să se rostogolească spre unul din capetele acesteia, la alegerea jucătorului, cu aceeaşi viteză de 1 metru pe secundă. Când mingea ajunge la capătul platformei, îşi continuă căderea liberă, pe verticală. Mingea nu are voie să cadă liber mai mult de MAX metri odată (între două platforme).
Scrieţi un program care determină un mod de rostogolire a mingii pe platforme astfel încât să ajungă la podea cât mai repede posibil. (CEOI România, 2000)

Date de intrare
Fişierul FALL.IN:
*Linia 1***:** N X Y MAX

- Patru numere întregi, separate prin câte un spaţiu: numărul de platforme (excluzând podeaua), poziţia de plecare a mingii (coordonatele pe orizontală şi verticală), şi distanţa maximă pe care are voie să cadă; platformele sunt numerotate de la 1 la N.

*Liniile 2..N+1***:** $X1_i$ $X2_i$ H_i

- Trei numere întregi, separate prin spaţii; platforma *i* este situată la înălţimea H_i, între coordonatele orizontale $X1_i$ şi $X2_i$ inclusiv ($X1_i<X2_i$, i=1..N).

Observaţii

– Diametrul mingii şi grosimea platformelor se ignoră. Dacă mingea cade exact pe marginea unei platforme, se consideră că a căzut pe platformă.
– Oricare două platforme nu au puncte comune.
– Pentru datele de test există întotdeauna soluţie.
– Toate dimensiunile sunt exprimate în metri.

Ieşire
Numele fişierului: FALL.OUT
Linia 1: TIME

- Un număr întreg, reprezentând timpul la care mingea atinge podeaua, conform soluţiei voastre.

Celelalte linii, până la sfârşitul fişierului:
P T D

- Trei numere întregi, separate prin spaţii. Mingea atinge platforma P la momentul T şi se îndreaptă în direcţia D (S pentru stânga şi D pentru dreapta).
- Impactul cu podeaua nu trebuie să apară în aceste linii.

- Impacturile cu platformele se precizează astfel încât valorile succesive ale lui T să fie în ordine crescătoare.

Observaţie

Dacă există mai multe soluţii posibile, se cere numai una.

Restricţii

- $1 \le N \le 1000$
- $-20000 \le X1_i, X2_i \le 20000, \forall i \in \{1,2,...,N\}$
- $0 < H_i < Y \le 20000$

Exemplu

FALL.IN	FALL.OUT
3 8 17 20	23
0 10 8	2 4 D
0 10 13	1 11 D
4 14 3	3 16 D

Soluţie

Vom reţine informaţiile despre platforme în 3 vectori (x1[], x2[], h[]). Pentru a uniformiza datele problemei vom introduce două platforme artificiale: platforma 0, reprezentată de punctul de plecare (x1[0]=x, x2[0]=x, h[0]=y) şi podeaua, platforma n+1 (x1[n+1]=-30000, x2[n+1]=30000, h[0]=0).

```
#define NMax 1003
int x1[NMax], x2[NMax], h[NMax];
int n, hmax;
void citire()
{ifstream fin("fall.in");
 int x, y;
 fin>>n>>x>>y>>hmax;
 for (int i=1; i<=n; i++)
     fi>>x1[i]>>x2[i]>>h[i];
 fi.close();
 x1[0]=x; x2[0]=x; h[0]=y;
 x1[n+1]=-30000; x2[n+1]=30000; h[n+1]=0;
 n+=2;
 fin.close();}
```

Sortez platformele în ordinea descrescătoare a înălţimilor lor. În vectorul p[] voi reţine indicii platformelor, în ordinea respectivă:

```
int p[NMax];
void sort()
{int sch, aux;
 for (int i=0; i<n; i++) p[i]=i;
 do {sch=0;
     for (i=2; i<n; i++)
         if (h[p[i]]>h[p[i-1]])
            {aux=p[i]; p[i]=p[i-1]; p[i-1]=aux; sch=1;}}
 while (sch); }
```

De asemenea, pentru fiecare platformă putem calcula indicele platformei pe care va cădea mingea în cazul în care se rostogoleşte prin capătul din stânga (memorat în vectorul st[]), respectiv indicele platformei pe care va cădea mingea în cazul în care se rostogoleşte prin capătul din dreapta (memorat în vectorul dr[]):

```
int st[NMax], dr[NMax];
void StangaDreapta()
```

```
{int i;
for (i=0; i<n; i++)
    {st[p[i]]=dr[p[i]]=0;
     for (j=i+1; h[p[i]]-h[p[j]]<=hmax && (!st[p[i]] || !dr[p[i]]);j++)
           {if (!st[p[i]] && x1[p[j]]<=x1[p[i]] && x1[p[i]]<=x2[p[j]])
                st[p[i]]=p[j];
            if (!dr[p[i]] && x1[p[j]]<=x2[p[i]] && x2[p[i]]<=x2[p[j]])
                dr[p[i]]=p[j];           }
     }
}
```

Utilizând aceste informaţii precalculate, vom aborda rezolvarea problemei prin metoda programării dinamice.

1. Subproblemele problemei date constau în determinarea timpului minim în care ajunge mingea pe podea în cazul în care cade prin marginea din stânga a platformei i, respectiv prin marginea din dreapta a platformei i, $\forall i \in \{0,1,...,n-1\}$.
2. Vom reţine soluţiile subproblemelor utilizând 4 vectori mst[], mdr[], undest[] şi undedr[], având următoarea semnificaţie:

– mst[i]=timpul minim în care mingea ajunge pe podea, dacă porneşte din marginea din stânga a platformei i;
– mdr[i]=timpul minim în care mingea ajunge pe podea, dacă porneşte din marginea din dreapta a platformei i;
– undest[i]=direcţia în care trebuie să se deplaseze mingea atunci când cade din marginea stângă a platformei i ('S' pentru stânga, 'D' pentru dreapta);
– undedr[i]=direcţia în care trebuie să se deplaseze mingea atunci când cade din marginea dreaptă a platformei i ('S' pentru stânga, 'D' pentru dreapta).

Evident, timpul minim de cădere va fi memorat în mst[0]=mdr[0].

3. Rezolvarea relaţiei de recurenţă în mod *bottom-up* o vom face parcurgând platformele de jos în sus:

```
long int mst[NMax], mdr[NMax];
char undest[NMax], undedr[NMax];
void calcMin()
{ int i, j, lgst, lgdr;
  mst[n-1]=mdr[n-1]=0;
  for (j=n-2; j>=0; j--)
      {i=p[j];  mst[i]=mdr[i]=Infinit;
       if (st[i]) //mingea cade prin stanga platformei i
          {
            if (st[i]==n-1) lgst=lgdr=0;
               else
               {lgst=mst[st[i]]+abs(x1[i]-x1[st[i]]);
                lgdr=mdr[st[i]]+abs(x1[i]-x2[st[i]]);}
            if (lgst<lgdr)
               {mst[i]=lgst+(h[i]-h[st[i]]);
                undest[i]='S'; }//cade pe st[i] spre stanga
                else
                {mst[i]=lgdr+(h[i]-h[st[i]]);
                 undest[i]='D';}
          }
       if (dr[i]) //mingea cade prin dreapta platformei i
          {
          if (dr[i]==n-1) lgst=lgdr=0;
             else
             {lgst=mst[dr[i]]+abs(x2[i]-x1[dr[i]]);
```

```
                lgdr=mdr[dr[i]]+abs(x2[i]-x2[dr[i]]);}
            if (lgst<lgdr)
               {mdr[i]=lgst+h[i]-h[dr[i]]; undedr[i]='S';}
                else
                {mdr[i]=lgdr+h[i]-h[dr[i]]; undedr[i]='D';}
            }
        }
}
```

4. Reconstituirea soluţiei o vom face utilizând informaţiile deja memorate:

```
void Afisare ()
{ ofstream fout("fall.out");
 long t; int  i, x; char d;
 fout<<mst[0]<<endl;
 for (x=x1[0], i=t=0, d='S'; i<n-2; )
     {//sunt pe platforma i si ma deplasez in directia d
     if (d=='S')
        { t+=h[i]-h[st[i]]+x-x1[i];
          x=x1[i]; d=undest[i];  i=st[i]; }
        else
        { t+=h[i]-h[dr[i]]+x2[i]-x;
          x=x2[i]; d=undedr[i]; i=dr[i];  }
     fout<<i<<' '<<t<<' '<<d<<endl;   }
 fout.close();}
```

Problema 6 Doipatru

https://infoarena.ro/pd

Membrii Lotului Naţional de Informatică sunt foarte mândri de noul joc inventat de ei, pe care l-au denumit asemănător cu o problemă de la Olimpiada Internaţională de Informatică din anul 2001, care le-a plăcut foarte mult. Astfel, jocul se numeşte DoiPatru.

Pentru acest joc se folosesc $3 \leq N \leq 30$ grămezi, fiecare conţinând cel puţin 0 şi cel mult 4 bile. Numărul total de bile din toate grămezile este *2*N*. Doi jucători mută alternativ. Atunci când îi vine rândul, fiecare jucător este obligat să efectueze o mutare validă.

O mutare validă constă din alegerea a două grămezi, dintre care prima grămadă are mai multe bile decât cea de a doua. Jucătorul ia o bilă din prima grămadă şi o mută în cealaltă. Mutarea se consideră validă, doar dacă numărul de bile rezultat în a doua grămadă după mutarea bilei nu este mai mare decât numărul de bile rămas în prima grămadă.

Jocul se termină atunci când nu mai poate fi efectuată nici o mutare validă (dacă vă gândiţi puţin, veţi constata că acest lucru se întâmplă atunci când fiecare grămadă conţine două bile).

Câştigătorul jocului este desemnat cel care deţine mai multe grămezi la sfârşitul jocului. Bineînţeles, dacă cei doi jucători deţin un număr egal de grămezi, jocul se consideră a fi remiză.

Un jucător deţine o grămadă dacă grămada are două bile, iar acest număr (de două bile) a rezultat în urma unei mutări efectuate de jucătorul respectiv. De exemplu, dacă un jucător alege o grămadă cu 4 bile şi una cu o bilă, în urma efectuării mutării, el va deţine cea de-a doua grămadă (care va avea două bile), dar prima nu va aparţine deocamdată nici unuia dintre jucători. Dacă alege o grămadă cu 3 bile şi una cu 0 bile, jucătorul va deveni proprietarul primei grămezi, deoarece, în urma mutării efectuate, grămada respectivă va rămâne cu două bile. În cazul în care alege o grămadă cu 3 bile şi una cu o bilă, după efectuarea mutării, el va deţine ambele grămezi (amândouă au acum doua bile).

Dacă un jucător este proprietarul unei grămezi la un moment dat în timpul jocului, nu înseamnă că această grămadă va rămâne în posesia lui până la sfârşit. De exemplu, să presupunem că jucătorul 1 deţine o grămadă cu două bile şi este rândul jucătorului 2 să mute. Dacă acesta alege o grămadă cu 4 bile şi grămada cu două bile ce aparţine jucătorului 1, după efectuarea mutării, ambele grămezi vor avea 3 bile, iar numărul de grămezi aflate în posesia jucătorului 1 va scădea cu 1 (grămada deţinută de el anterior nu mai aparţine nici unuia din cei doi jucători, căci nu mai are doua bile).

Dacă la începutul jocului există unele grămezi având două bile, acestea sunt distribuite în mod egal celor doi jucători. Dacă numărul de grămezi cu două bile este impar, atunci jucătorul 2 va primi cu o grămadă mai mult decât jucatorul 1. Jucătorul 1 este cel care efectuează prima mutare.

Scrieţi un program care, pentru un *N* dat şi un set de configuraţii iniţiale ale jocului cu *N* grămezi, decide rezultatul fiecărei configuraţii de joc.

Rezolvare:

Începem rezolvarea cu observaţia că există 5 tipuri de grămezi (0, 1, 2, 3 şi 4 bile), dar pentru fiecare grămadă de 2 bile trebuie să ştim cărui jucător aparţine. Deci, vom separa deci toate grămezile de 2 bile în 2 categorii: 2A care reprezintă grămezile de 2 bile deţinute de primul jucător şi 2B care sunt grămezile de 2 bile deţinute de al doilea jucător.

Orice moment al jocului poate fi reprezentat identificând jucătorul care urmează la mutare şi numărul de bile din fiecare dintre cele N grămezi. O variantă alternativă este de a reprezenta numerele de grămezi din

fiecare tip, starea fiind reprezentată prin valorile (J, n_0, n_1, n_{2A}, n_{2B}, n_3, n_4), unde n_k reprezintă numărul de grămezi care au k bile (cu excepţiile 2A şi 2B, descrise înainte).

Vom nota prin `R[J, n0, n1, n2A, n2B, n3, n4]` cel mai bun rezultat pe care îl poate obţine jucătorul care urmează (J). Valorile posibile ale rezultatului vor fi alese ca numere întregi crescătoare, astfel: 0 dacă din configuraţia curentă jucătorul nu are nici o şansă să câştige, 1 dacă jucătorul poate obţine o remiză şi 2 dacă jucătorul are o strategie sigură de câştig. Fiecare jucător va alege mutarea potrivită astfel încât să maximizeze valoarea R, care este în acelaşi timp rezultatul cel mai prost pentru celălalt jucător. Vom nota jucătorii prin 0 şi 1 şi putem calcula valoarea optimă pentru R astfel:

$$R[J, n_0, n_1, n_{2A}, n_{2B}, n_3, n_4] = 2 - \min\{R[1-J, n'_0, n'_1, n'_{2A}, n'_{2B}, n'_3, n'_4]\}$$

dacă (n'_0, n'_1, n'_{2A}, n'_{2B}, n'_3, n'_4) reprezintă o stare accesibilă din starea curentă printr-o singură mutare. Observăm că rezultatul este cu atât mai bun pentru unul dintre jucători cu cât este mai prost pentru celălalt, deci rezultatele sunt invers proporţionale, 2 pentru J reprezentând 0 pentru jucătorul 1 - J. Fiecare jucător alege mutarea care îi va obţine rezultatul maxim, acesta fiind corespunzător rezultatului defavorabil (de valoare minimă) pentru celălalt.

Stările pentru care nu mai există decât grămezi de tipuri 2A şi 2B sunt, conform enunţului, stări finale.

$$R[J, 0, 0, n_{2A}, n_{2B}, 0, 0] = \begin{cases} 0 & J = 0, n_{2A} < n_{2B} \\ 0 & J = 1, n_{2A} > n_{2B} \\ 1 & n_{2A} = n_{2B} \\ 2 & J = 0, n_{2A} > n_{2B} \\ 2 & J = 1, n_{2A} < n_{2B} \end{cases}$$

Notând cu S tuplul distribuţiei bilelor în grămezi (J, n_0, n_1, n_{2A}, n_{2B}, n_3, n_4) atunci vom folosi o notaţie echivalentă dar mai scurtă, R[J, S]. Vom iniţializa toate valorile din acest tablou multidimensional cu -1 şi apoi vom calcula recursiv valorile, obţinând o complexitate polinomială prin memoizare.

```
calculR(J, S)
dacă R[J, S] != -1 atunci
     returnează R[J, S];
dacă S e stare finală
     returnează 0, 1 sau 2 în funcţie de câştigător;
R[J, S] = 0;
pentru toate stările S' în care se poate ajunge din S printr-o mutare
      R[J, S] = max(R[J, S], 2 - calculR(1 - J, S'));
returnează R[J, S];
```

La prima vedere, numărul stărilor este $2 * (2N)^6 = 2 * 60^6 = 93{,}312 * 10^9$, deci numărul stărilor este prea mare pentru a intra în limite rezonabile de spaţiu şi timp. Observăm totuşi că toate stările îndeplinesc condiţia $n_0 + n_1 + n_{2A} + n_{2B} + n_3 + n_4 = N$. Numărul total al stărilor este numărul tuplurilor care îndeplinesc egalitatea şi condiţiile $n_k \geq 0$. Vom numerota cele 6 tipuri de grămezi prin numere de la 0 la 5. Vom calcula valorile $N_S[g, b, v]$ care reprezintă numărul de variante de a grupa b bile în g grămezi, astfel încât cea mai mică grămadă să aibă cel puţin mărimea tipului de indice $0 \leq v \leq 5$. Notăm cu D şirul dimensiunilor tipurilor, (0, 1, 2, 2, 3, 4) şi observăm că o stare descrisă prin (g, b, v) ori are o grămadă de dimensiune D_v ori toate grămezile sunt mai mari decât această dimensiune. Atunci recurenţa de calcul

pentru N_S este: $N_S[g, b, v] = N_S[g, b, v + 1] + N_S[g - 1, b - D_v, v]$ cu cazul particular $N_S[0, 0, 5] = 1$. Numărul total de stări este valoarea $N_S[N, 2*N, 0]$. Pentru valoarea maximă a lui N din enunţ, am obţinut $N_S(30) = N_S[30, 60, 0] = 8266$. Rezultă deci că numărul total real al stărilor este relativ mic, iar soluţia noastră se încadrează în limitele de timp şi spaţiu date.

Pentru o stare S dată, numărul stărilor posibile S' în care putem ajunge este foarte mic. În cel mai rău caz, perechile de grămezi care reprezintă mutarea din pasul curent sunt din mulţimea {(0, 2A), (0, 2B), (0, 3), (0, 4), (1, 3), (1, 4), (2A, 4), (2B, 4)}, deci un jucător are maxim 8 mutări posibile de efectuat. După ce am ales tipurile de grămezi pe care efectuăm mutarea, este irelevantă alegerea grămezilor cu dimensiunile date, deoarece toate alegerile duc la aceeaşi stare următoare.

Un mod de a stoca tabloul R astfel încât spaţiul necesar să fie minim este folosirea unui dicţionar care să stocheze valorile $R[J, n_0, n_1, n_{2A}, n_{2B}, n_3, n_4]$ pentru toate stările valide, folosind astfel doar $O(N_S(N))$ spaţiu. Dicţionarul poate fi implementat printr-o tabelă de dispersie sau arbori binari de căutare, în C++ folosind chiar map sau hash_map din STL. Altă opţiune este codificarea fiecărei stări S printr-un număr întreg cuprins între 1 şi $N_S(N)$, caz în care tabloul R poate fi stocat într-o matrice de dimensiune $2xN_S(N)$. Asocierea dintre descrierea unei stări (numărul de grămezi de fiecare tip) şi numărul său de ordine poate fi precalculată şi stocată într-un dicţionar sau poate fi calculată în O(N) cu ajutorul unor formule. În primul caz, vom genera prin backtracking toate variantele posibile de stări într-o ordine oarecare şi vom aloca fiecărei stări câte un număr, stocând izomorfismul într-un dicţionar.

Varianta mai complexă, dar mai elegantă presupune stabilirea unei ordini fixe a stărilor şi apoi folosirea unor tehnici combinatoriale pentru a calcula numărul corespunzător unei stări sau starea corespunzătoare unui număr. Vom defini mai formal o stare ca un 6-tuplu $(n_0, n_1, n_{2A}, n_{2B}, n_3, n_4)$ şi vom ordona toate stările lexicografic (stările sunt ordonate după n_0; în caz de egalitate, sortarea se face după n_1 ş.a.m.d.). Atunci numărul de ordine al unei stări calculat de formula de mai sus este numărul de stări care au o valoare mai mică pentru n_0 plus numărul de stări care au aceeaşi valoare pentru n_0 dar o valoare mai mică pentru n_1 ş.a.m.d.

```
IS = 1;
G = N;
B = 2 * N;
pentru k = 0, 4 execută
        dacă n[k] > 0 atunci
             pentru i = 1, n[k] execută
                    IS = IS + S[G][B][k + 1];
                    G = G - 1;
                    B = B - D[k];
              sfârşit pentru;
  sfârşit pentru;
returnează IS;
```

Operaţia inversă calculează o stare S pe baza valorii I(S) a indicelui stării.

```
G = N;
B = 2 * N;
pentru k = 0, 4 execută
      n[k] = 0;
      cât timp IS > S[G][B][k + 1] execută
                IS = IS - S[G][B][k + 1];
                G = G - 1;
                B = B - D[k];
                n[k] = n[k] + 1;
      sfârşit cât timp;
sfârşit pentru;
n[5] = G;
returnează n;
```